VIPER

December 1981-January 1982

Volume 3, Number 5 Journal of the VIP Hobby Computer Assn.
The VIPER was founded by ARESCO, Inc. in June 1978

****

Contents

The VIPER, founded by ARESCO, Inc., in July 1978, is the Official Journal of the VIP Hobby Computer Association. Acknowledgement and appreciation is extended to ARESCO for permission to use the VIPER name. The Association is composed of people interested in the VIP and computers using the 1802 microprocessor. The Association was founded by Raymond C. Sills and created by a Constitution, with By-Laws to govern the operation of the Association. Mr. Sills is serving as Director of the Assocation, as well as editor and publisher of the VIPER.

The VIPER will be published six times per year and sent to all members in good standing. Issues of the VIPER will not carry over from one volume to another. Individual copies of the VIPER and past issues, where they are available, may be sent to interested people for $3 each. Annual dues to the Association, which includes six issues of the VIPER, is $12 per year.

VIP and COSMAC are registered trademarks of RCA Corporation. The VIP Hobby Computer Associatin is in no way associated with RCA, and RCA is not responsible for the contents of this newsletter. Members should not contact RCA regarding material in the VIPER. Please send all inquiries to VIPHCA 32 Ainsworth Avenue, East Brunswick, NJ 08816.

Membership in the VIP Hobby Computer Association is open to all people who desire to promote and enjoy the VIP and other 1802 based systems. Send a check for $12 payable to "VIP Hobby Computer Association" c/o R. C. Sills at the above address. People outside of the United Stated, Canada or Mexico please include $6 extra for postage. All funds must be in U>S. Dollars.

Contributions by members or interested people are welcome at any time. Material submitted by you is assumed to be free of copyright restrictions and will be considered for publication in the VIPER. Articles, letters, programs, etc., in camera-ready form on 8.5 by 11 inch paper will be given preferential consideration. Please send enough information about any program so that the readers can operate the program. Fully documented programs are best, but "memory dumps" are OK if you provide enough information to run the program. Please indicate in your material any key memory locations and data areas.

ADVERTISING RATES

1. Non-commercial classified ads from members, 5 cents per word, minimum of 20 words. Your address or phone number is free.

2. Commercial ads and ads from non-members, 10 cents per work, minimum of 20 words. Your address or phone number is free.

3. Display ads from camera ready copy, $6/half page, $10/page.

Payment in full must accompany all ads. Rates are subject to change.

If you write to VIPER/VIPHCA, please indicate that it is OK to print your address in your letters to the editor, if you want that information released. Otherwise, we will not print your address in the VIPER.

STEP BY STEP
A Single Stepper/Trace Debugger for your Chip-8 Interpreter

by Tom Swan

The bugs we have in Mexico would possibly seem strange to you. There are the usual selections of scorpions and huge spiders capable of weaving webs of steel to catch their prey. (True--I've tried without success to break only a double strand from such a web.) They have a horrible thing called a Vinagria that emits a vinegarlike smell and has a deadly bite. And some of the hairy mygalomorphs (Tarantulas) even feast on small birds and lizards.

But the bugs in my Chip-8 programs are all of the universal type. (You knew that was coming, didn't you?) One way to catch such bugs (without a net of course) is to insert break points in your program, run the devil up to that spot, then check the variables using the operating system to find them in memory. Another way is to use the following Single Stepper program to execute only one instruction at a time with the option of viewing all Chip-8 variables on your display while the program is running. In addition, a trace feature displays the address of the next instruction to be executed so you can follow program execution flow.

This debug program works with any Chip-8 program and may be incorporated into any interpreter whether or not it contains some of the modifications suggested in past VIPERS. Since Chip-8 is graphics oriented (as opposed to an interpreter designed to manipulate numbers or text), you have the option of single stepping through a program while watching the results on display, or viewing variables without disturbing the contents of the Chip-8 display.

To run the program, you need to have an external keyboard hooked up to the VIP's input port with its strobe line connected to the EF-4 flag line. (If you don't have a keyboard you can still use this program -- see end of article!)

Next, study the accompanying memory map to decide where to put the Single Stepper program. If you have 4K of memory, I suggest you load the program into page 0E00 where the Chip-8 variables, work space and stack are located. 2K systems could use 0600. You may load the Single Stepper into any memory page beginning with the first byte of that page as the Single Stepper is page relocatable. Naturally you cannot use the same memory area for Chip-8 programs anymore.

The Single Stepper also requires its own display page which will automatically be located two pages above whichever page is being used for displays by the Chip-8 interpreter. Therefore if you have a 4K system, the debugging display would be located at 0D00 for most programs and your program must not exceed 0CFF.

A 2K system would require 0500-07FF total, meaning you may not program beyond 04FF.

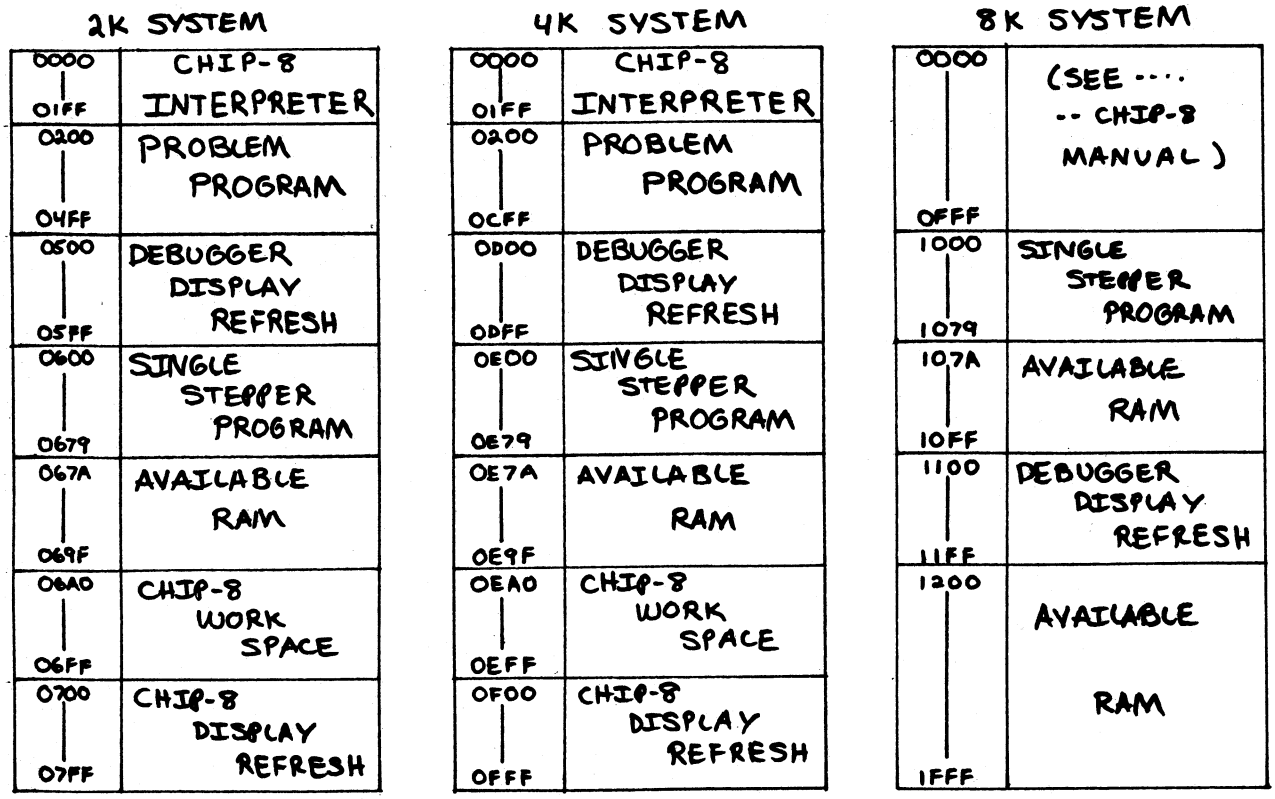
LOADING AND RUNNING

After deciding where you want the Single Stepper to go, load it in and record one page (from wherever you loaded the program) to save it for future debugging. Load your normal (or other) Chip-8 interpreter and your problem program just as usual. You may run the program subject to the memory restrictions described without disturbing the Single Stepper.

To call the Single Stepper into action, enter the following change to your Chip-8 interpreter after of course loading the Single Stepper into memory.

```
001B  F8  LDI      ;Load address of Single Stepper
  1C  XX               (Page address of program goes here)
  1D  B3  PHI   R3 ;Put in R3.1
  1E  F8  LDI      ;Load 00 into D register
  1F  00

0020  A3  PLO   R3 ;Put in R3.0
  21  D3  SEP   R3 ;Call debugger
```

MEMORY MAP



At 001C you must enter the page address of the memory area where you installed the Single Stepper. If you entered it at 0E00, then program an 0E byte at 001C. If you put it at 0600, then an 06 byte goes in 001C.

Flip the run switch up. Nothing happens, right? Press the space bar on your keyboard several times until you hear a beep. Now press Key D to activate the variables/trace display option. Repeated presses of the space bar single steps your Chip-8 program one instruction at a time. Pressing Key D calls the display option back again and you must always press the space bar to continue stepping through your program. Pressing Key D twice in a row has no useful effect.

WHAT YOU SEE IS WHAT YOU GET

All Chip-8 variables V0-VF are displayed on the screen starting in the upper left corner and reading across the page from left to right. Each variable, remember, is a pair of the hex digits so there are four rows of four variables displayed. The last four digits displayed tell you the address of the next Chip-8 instruction to be executed -- allowing you to trace a program through memory. I suggest you load a Chip-8 program that you know works (the figure shooting at moving target game is a good choice) until you get the feel of operating the debugger. You have to single step through a lot of instructions till you see something happening, but keep hitting the space bar -- something eventually will.

OPTIONS

Of course a disadvantage of this program is that it takes away some Chip-8 programming space. If you have external memory, I suggest you enter the debugger somewhere higher than 0FFF, the highest addressable Chip-8 memory location. You may also want to locate the debugger's display on external RAM. This is easily done by adjusting RB.1 to the page address where you want the display to go. The following change will do this:

```
0X21  F8  LDI      ;Load page address
  22  XX           ;of debug display
```

Make this change to the Single Stepper program and enter the page address at 0X22 where you want the display to go. If you entered the Single Stepper at 1000-1079 for example, you may want the display to reside in the memory page at 1100-11FF. You would enter F811 at 1021 to set that up and you would call the program from the interpreter as described before using "10" as the page address at location 001C. You may use this same technique to fix the debugging display at

a known location if your Chip-8 program will change its normal display area -- the highest on card RAM page in your system. Remember that the Single Stepper uses the display page two pages above (lower address) than the one currently used by your program. Your display page is correctly reset, however, no matter where it is by the Single Stepper after you press the space bar.

You may use the Single Stepper with the special two-page interpreter supplied with my book Pips for Vips. A two-page display results in a squished debug display though the Single Stepper will only use one page even though you are displaying two. The bottom half of the display area is still usable for programs and won't be affected by the debugger. You must change 0211 to FF however to allow off card RAM displays -- this won't affect programs and may be left in as a permanent change to the two-page interpreter.

One problem is that the tone generator will no longer work while using the Single Stepper. You may reactivate tone generation by making the following change to the debugger program:

```
0X08  E2  NOP       ;Cancel keyboard tone
```

Now your keyboard will operate silently. If you have an external speaker on your keyboard, you will probably want to make this change as a permanent addition to the debugger.

Another problem is that timing loops will have no meaning since your program is no longer operating at speed, and the Chip-8 timer (R8.1 in the interrupt routine) is still working full time. You may still view the effect of a timing loop though it will not probably loop the same number of times as it will when your program runs normally.

Random number generation is still valid and presumably equivalent to normal execution conditions though I may be making an invalid assumption about the random number generator with that statement. Any effect is likely to be very small, however, and totally nonconsequential to most game programs.

If you care to inspect the workings of the debugger you will see that it duplicates the display routines in ROM (using the bit tables though) rather than taking a suggestion from an early VIPER to gain access to these routines. This was done to preserve RA, the Chip-8 memory pointer which is changed by the hex digit display routine in ROM.

I don't propose running entire programs by single stepping to see if they work. That would be somewhat akin to turning the propeller of an airplane by hand to test if the thing flies. As a subroutine developer tool, the debugger should be very useful when you're not sure just how the variables

got all mixed up and you want to slow things down to a viewable rate.

You can also beat the pants off of Jackpot by holding down the appropriate hex pad keys when the right symbols appear. Jackpot every time! Of course it takes two weeks to play a game, but ...

Have fun.

SINGLE STEP AND TRACE FOR CHIP-8 -- RUNS IN R3

```
0X00  E2  SEX   2   ;X=2
  01  22  DEC   R2  ;Decrement stack pointer
  02  9B  GHI   RB
  03  BF  PHI   RF  ;Save display page in RF.1
  04  3F  BN4       ;Loop here till key is pressed (A)
  05  04                (on external keyboard)
  06  F8  LDI       ;Sound tone while key is
  07  01                held down
  08  A8  PLO   R8
  09  37  B4        ;Loop until key is released
  0A  06                (tone goes off automatically)
  0B  6B  INP       ;Get input byte from keyboard
  0C  FB  XRI       ;Test if = hex value for space bar
  0D  20
  0E  3A  BNZ       ;If not, branch to next part
  0F  1B

0X10  9F  GHI   RF  ;Reset RB to old display page (which is
  11  BB  PHI   RB       not affected by this routine)
  12  12  INC   R2  ;Reset stack pointer
  13  96  GHI   R6  ;R7.1=R6.1 - begin duplication
  14  B7  PHI   R7  ;      of patched instructions
  15  F8  LDI       ;RC.1=00
  16  00
  17  BC  PHI   RC
  18  45  LDA   R5  ;Get first half of a Chip-8 instruction
  19  AF  PLO   RF  ;Put in RF.0 (also in "D" on return)
  1A  D4  SEP   R4  ;Return from single step
  1B  F8  LDI       ;Test if Key D pressed
  1C  44
  1D  F3  XOR       ;By comparing with hex 44
  1E  3A  BNZ       ;If ≠ D, then not space bar either
  1F  04            ;Branch for a valid entry

0X20  9F  GHI   RF  ;Switch to a display page
  21  FF  SMI       ;Two memory pages back from
  22  02            ;Present one -- see notes to
  23  BB  PHI   RB  ;Alter this if you have 8K or more
                        in your system
  24  BE  PHI   RE  ;Also put this value in RE.1
  25  F8  LDI       ;Set RE.0 = FF which will point to
```

```
0X26  FF            ;Bottom of the one page display area
  27  AE  PLO   RE  ;
  28  F8  LDI       ;Erase display page from bottom up
  29  00            ;       "               "
  2A  5E  STR   RE  ;       "               "
  2B  8E  GLO   RE  ;       "               "
  2C  2E  DEC   RE  ;       "               "
  2D  3A  BNZ       ;       "               "
  2E  28
  2F  1E  INC   RE  ;RE + 1 = top of display page again

0X30  F8  LDI       ;RC.1 = 81 which will address the
  31  81            ;Display bits in VIP ROM space
  32  BC  PHI   RC
  33  93  GHI   R3  ;Set up RD for use as a subroutine
  34  BD  PHI   RD  ;Program counter for display two
  35  F8  LDI       ;Digits sub
  36  51
  37  AD  PLO   RD
  38  F8  LDI       ;Set R6 to first Chip-8 variable
  39  F0            ;Stored at 0YF0-0YFF
  3A  A6  PLO   R6
  3B  46  LDA   R6  ;Get a variable
  3C  DD  SEP   RD  ;Do sub -- Display 2 Digits
  3D  8E  GLO   RE  ;Test the cursor (display destination
                        address)
  3E  FA  ANI       ;"AND" with "07" to see if carriage
  3F  07                return is needed

0X40  3A  BNZ       ;If not (i.e. last 4 bits RE ≠ 0) then
  41  46                branch to continue
  42  8E  GLO   RE  ;Else add 28 hex (40 decimal) to
  43  FC  ADI       ;Bring the cursor down a row
  44  28
  45  AE  PLO   RE
  46  86  GLO   R6  ;Test variable pointer R6
  47  3A  BNZ       ;If ≠ 0, then branch back to continue
  48  3B            ;(When R6.0 = 0, then 16 variables
                        are shown)
  49  26  DEC   R6  ;Reset R6.1 to previous value
  4A  95  GHI   R5  ;Display high part of R5 (Chip-8 PC)
  4B  DD  SEP   RD  ;Do sub - Display 2 Digits
  4C  85  GLO   R5  ;Display low part of R5
  4D  DD  SEP   RD  ;Do sub - Display 2 Digits
  4E  30  BR        ;Branch to point A to wait next user
  4F  04            ;Command


                    DISPLAY TWO DIGITS SUB

0X50  D3  SEP   R3  ;Return leaving RD at next address
  51  AC  PLO   RC  ;Save D in RC.0 for the moment
  52  FA  ANI       ;Strip all but last 4 bits from the
  53  0F            ;Hex byte to be displayed
```

```
0X54  73  STXD      ;Push this value onto the stack
  55  8C  GLO   RC ;Get the same hex value from RC.0
  56  F6  SHR      ;Shift 4 times to the right
  57  F6  SHR      ;To move the highest digit to
  58  F6  SHR      ;The lowest position
  59  F6  SHR
  5A  73  STXD     ;Push this onto the stack
  5B  F8  LDI      ;Set up R7.0 as a loop counter of 2
  5C  02
  5D  A7  PLO   R7
  5E  60  IRX      ;Point to saved data on stack
  5F  F0  LDX      ;Pop data off stack (leaving RX where
                        it was)
0X60  AC  PLO   RC ;Put in RC.0 to address bits table in ROM
  61  0C  LDN   RC ;Get value from table
  62  AC  PLO   RC ;Put in RC.0 to address bits for this digit
  63  F8  LDI      ;Set up RF.0 as a loop counter of 5
  64  05           ;To display entire digits
  65  AF  PLO   RF ;
  66  4C  LDA   RC ;Get a bit pattern byte
  67  5E  STR   RE ;Store @ cursor RE in display area
  68  8E  GLO   RE ;Move cursor down one byte row (of 8)
  69  FC  ADI      ;To continue displaying the digit
  6A  08
  6B  AE  PLO   RE
  6C  2F  DEC   RF ;Decrement byte loop counter
  6D  8F  GLO   RF ;Test count
  6E  3A  BNZ      ;If ≠ 0, branch till one digit displayed
  6F  66

0X70  8E  GLO   RE ;Reset cursor RE so that next
  71  FF  SMI      ;Digit to be displayed will
  72  27           ;Appear to the right of the present one
  73  AE  PLO   RE
  74  27  DEC   R7 ;Decrement the digit loop counter
  75  87  GLO   R7 ;Test the count
  76  3A  BNZ      ;Branch if ≠ 0 to display two digits
  77  5E
  78  30  BR       ;Else branch to exit
  79  50
```

FLASH! -- Nonkeyboard owners please note:

You may still use this program even if you do not own a keyboard. The following changes to the Single Stepper place timing loops where the keyboard sensing routines used to be. While you have no control over the stepping process, it will be slow enough to help you debug your problem program and check out variables during execution.

In addition to the Single Stepper as listed, enter the following:

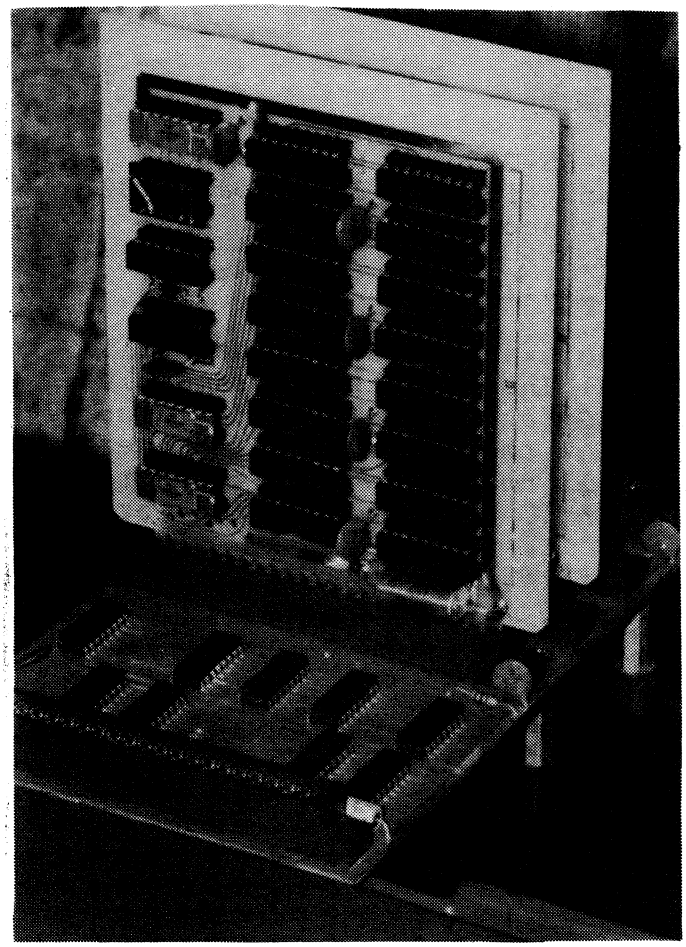
```
0X04   F8   LDI       ;Load timer value into RE.1
  05   A0             ;(Interrupt timer must not be used)
  06   BE   PHI   RE  ;(As TV is off first time  through here)
  07   2E   DEC   RE  ;Decrement timer
  08   9E   GHI   RE  ;Test timer
  09   3A   BNZ       ;If ≠ 0, loop to waste time here
  0A   07
  0B   30   BR        ;Then branch to debug
  0C   20
  --

0X4F   7A             ;Branch to 0X7A when done with display
  --

0X7A   F8   LDI       ;Identical timer as above -- value
  7B   A0             ;May be different here than in 0X05
  7C   BE   PHI   RE
  7D   2E   DEC   RE
  7E   9E   GHI   RE
  7F   3A   BNZ

0X80   7D
  81   30   BR        ;When done, branch to exit
  82   10             ;At 0X10
```

You may adjust the timing values to slow down or speed up the debugger by trying new values at 0X05 and 0X7B. (If you only want to step through a program and do not want to see the variables display, change 0X0C to hex 10.) Your tone generator will work -- do not make the suggested modification for tone generation discussed earlier!

When you flip to run, nothing will happen for several seconds while the interpreter goes through its initialization routines and turns on the display. Then you will begin to see your program running at tortoiselike speed, flipping back and forth from the Chip-8 display to the debug display automatically.

NOTE: Sometimes you may wish to begin single stepping somewhere other than at the beginning of a CHIP-8 program. In that case, enter a jump instruction at $0200 to go to where you want to start the debugging process.

8K RAM BOARD FOR YOUR RCA VIP



*USES POPULAR LOW-POWER 2114 RAMS

*EACH 4K BLOCK SEPARATELY ADDRESSABLE

*G10 GLASS BOARD

*DOUBLE SIDED 2 OZ COPPER

*SOLDER PLATED

*GOLD PLATED CONTACTS

*4-1/2" W x 5-3/8" H

*PLUGS INTO VIP EXPANSION CONNECTOR OR SYSTEM EXPANSION BOARD

*DRAWS APPROXIMATELY 600 mA FROM +5 V LINE

*BARE BOARD WITH DATA
$ 49.00.....POSTAGE PAID

*ASSEMBLED AND TESTED
$ 149.00.....POSTAGE PAID
CALIFORNIA RESIDENTS
ADD STATE SALES TAX

THE GJK 8K RAM CARD ALLOWS YOU TO ECONOMICALLY EXPAND YOUR VIP TO FULL CAPACITY (32K) WITHIN THE CONFINES OF THE RCA VP-575 SYSTEM EXPANSION BOARD.

COMING SOON:
12K 2716 EPROM CARD

SEND CHECK TO MONEY ORDER TO:

G. J. KRIZEK
722 N. MORADA AVENUE
WEST COVINA, CA 91790

3.05.10

FOR SALE: Basic Quest Super ELF with Hi & Lo address displays and some software. Good condition. $60.00 phone: 412-372-8773
Jeff Jones, 507 Cherry Dr., Trafford, PA 15085

READER I/O

Has anyone interfaced the VIP to CRT via a RS-232 I/O port using a mini monitor similar to the one in Microcomputing May 1980, page 132? I would be interested in hearing how it was accomplished.
John T. Keys, 102 Elm Street, Vienna, VA 22160
::

I hope to be submitting programs soon on software/hardware that will interface the Texas Instruments and General Instruments programmable sound chips (SN76489A/94 and AY-3-8910/8912 respectively.) I am also working on the hardware mods that will allow the color board to operate at maximum resolution (8 by 128), but I will need help with thesoft-software, if anyone is interested in helping. Keep up the good work with the VIPER.
Jeff Jones, 507 Cherry Dr., Trafford, PA 15085

FOR SALE: RCA VIP Tiny BASIC in ROM, $25.00. GRI 53 key ASCII keyboard (no enclosure) 5V only $55.00, includes an interface board for VIP. Both for only $75 and I pay postage.
Larry Dolce, Box J, Valley Cottage, NY 10989

REMINDERS

There are still a number of back issues of VIPER available: Partial sets of Vol. 1, and complete sets of Vol. 2. The price for Vol. 1 is $5, and Vol 2 is $8, postage paid for U.S. addresses. Send your check to VIP Hobby Computer Assn. at the usual address.

The VIP Ham net, Tuesdays (9PM EST on 3860KHz) with K2ULR, K2ZLU, AE1I, W20C and others. If there is interest, we might have a net on 15 or 20 Meters for members not on the East Coast. Let me know if you are interested. (RS)

CHIP-8 for ELF

by

Leo F. Hood

This program uses the VIP operating system for ELF program (VIPER 3.04.07)
od. Nov-81-

```
0000  C01000    LBR   #1000
0004  B2        PHI   R2
0005  B6        PHI   R6
0006  F8CF      LDI   #CF
0008  A2        PLO   R2
0009  F811      LDI   #11
000B  B1        PHI   R1
000C  F846      LDI   #46
000E  A1        PLO   R1
000F  90        GHI   R0
0010  B4        PHI   R4
0011  F81B      LDI   #1B
0013  A4        PLO   R4
0014  F801      LDI   #01
0016  B5        PHI   R5
0017  F8FC      LDI   #FC
0019  A5        PLO   R5
001A  D4        SEP   R4
001B  96        GHI   R6
001C  B7        PHI   R7
001D  E2        SEX   R2
001E  94        GHI   R4
001F  BC        PHI   RC
0020  45        LDA   R5
0021  AF        PLO   RF
0022  F6        SHR
0023  F6        SHR
0024  F6        SHR
0025  F6        SHR
0026  3244      BZ    #44
0028  F950      ORI   #50
002A  AC        PLO   RC
002B  8F        GLO   RF
002C  FA0F      ANI   #0F
002E  F9F0      ORI   #F0
0030  A6        PLO   R6
0031  05        LDN   R5
0032  F6        SHR
0033  F6        SHR
0034  F6        SHR
0035  F6        SHR
0036  F9F0      ORI   #F0
0038  A7        PLO   R7
0039  4C        LDA   RC
003A  B3        PHI   R3
003B  8C        GLO   RC
003C  FC0F      ADI   #0F
003E  AC        PLO   RC
003F  0C        LDN   RC
0040  A3        PLO   R3
0041  D3        SEP   R3
0042  301B      BR    #1B
0044  8F        GLO   RF
0045  FA0F      ANI   #0F
0047  B3        PHI   R3
0048  45        LDA   R5
0049  3040      BR    #40
004B  22        DEC   R2
004C  69        INP    1
004D  12        INC   R2
004E  D4        SEP   R4
004F  00        IDL
0050  00        IDL
0051  01        LDN   R1
0052  01        LDN   R1
0053  01        LDN   R1
0054  01        LDN   R1
0055  01        LDN   R1
0056  01        LDN   R1
0057  01        LDN   R1
0058  01        LDN   R1
0059  01        LDN   R1
005A  01        LDN   R1
005B  01        LDN   R1
005C  01        LDN   R1
005D  00        IDL
005E  01        LDN   R1
005F  01        LDN   R1
0060  00        IDL
0061  7C75      ADCI  #75
0063  83        GLO   R3
0064  8B        GLO   RB
0065  95        GHI   R5
0066  B4        PHI   R4
0067  B7        PHI   R7
0068  BC        PHI   RC
0069  91        GHI   R1
006A  EB        SEX   RB
006B  A4        PLO   R4
006C  D9        SEP   R9
006D  70        RET
006E  99        GHI   R9
006F  05        LDN   R5
0070  06        LDN   R6
0071  FA07      ANI   #07
0073  BE        PHI   RE
0074  06        LDN   R6
0075  FA3F      ANI   #3F
0077  F6        SHR
0078  F6        SHR
0079  F6        SHR
007A  22        DEC   R2
007B  52        STR   R2
007C  07        LDN   R7
007D  FA1F      ANI   #1F
007F  FE        SHL
```

```
0080  FE          SHL
0081  FE          SHL
0082  F1          OR
0083  AC          PLO   RC
0084  9B          GHI   RB
0085  BC          PHI   RC
0086  45          LDA   R5
0087  FA0F        ANI   #0F
0089  AD          PLO   RD
008A  A7          PLO   R7
008B  F8D0        LDI   #D0
008D  A6          PLO   R6
008E  93          GHI   R3
008F  AF          PLO   RF
0090  87          GLO   R7
0091  32F3        BZ    #F3
0093  27          DEC   R7
0094  4A          LDA   RA
0095  BD          PHI   RD
0096  9E          GHI   RE
0097  AE          PLO   RE
0098  8E          GLO   RE
0099  32A4        BZ    #A4
009B  9D          GHI   RD
009C  F6          SHR
009D  BD          PHI   RD
009E  8F          GLO   RF
009F  76          SHRC
00A0  AF          PLO   RF
00A1  2E          DEC   RE
00A2  3098        BR    #98
00A4  9D          GHI   RD
00A5  56          STR   R6
00A6  16          INC   R6
00A7  8F          GLO   RF
00A8  56          STR   R6
00A9  16          INC   R6
00AA  308E        BR    #8E
00AC  00          IDL
00AD  EC          SEX   RC
00AE  F8D0        LDI   #D0
00B0  A6          PLO   R6
00B1  93          GHI   R3
00B2  A7          PLO   R7
00B3  8D          GLO   RD
00B4  32D9        BZ    #D9
00B6  06          LDN   R6
00B7  F2          AND
00B8  2D          DEC   RD
00B9  32BE        BZ    #BE
00BB  F801        LDI   #01
00BD  A7          PLO   R7
00BE  46          LDA   R6
00BF  F3          XOR

00C0  5C          STR   RC
00C1  02          LDN   R2
00C2  FB07        XRI   #07
00C4  32D2        BZ    #D2
00C6  1C          INC   RC
00C7  06          LDN   R6
00C8  F2          AND
00C9  32CE        BZ    #CE
00CB  F801        LDI   #01
00CD  A7          PLO   R7
00CE  06          LDN   R6
00CF  F3          XOR
00D0  5C          STR   RC
00D1  2C          DEC   RC
00D2  16          INC   R6
00D3  8C          GLO   RC
00D4  FC08        ADI   #08
00D6  AC          PLO   RC
00D7  3BB3        BNF   #B3
00D9  F8FF        LDI   #FF
00DB  A6          PLO   R6
00DC  87          GLO   R7
00DD  56          STR   R6
00DE  12          INC   R2
00DF  D4          SEP   R4
00E0  9B          GHI   RB
00E1  BF          PHI   RF
00E2  F8FF        LDI   #FF
00E4  AF          PLO   RF
00E5  93          GHI   R3
00E6  5F          STR   RF
00E7  8F          GLO   RF
00E8  32DF        BZ    #DF
00EA  2F          DEC   RF
00EB  30E5        BR    #E5
00ED  00          IDL
00EE  42          LDA   R2
00EF  B5          PHI   R5
00F0  42          LDA   R2
00F1  A5          PLO   R5
00F2  D4          SEP   R4
00F3  8D          GLO   RD
00F4  A7          PLO   R7
00F5  87          GLO   R7
00F6  32AC        BZ    #AC
00F8  2A          DEC   RA
00F9  27          DEC   R7
00FA  30F5        BR    #F5
00FC  00          IDL
00FD  00          IDL
00FE  00          IDL
00FF  00          IDL
```

```
0100  FA0F      ANI   #0F
0102  308C      BR    #8C
0104  00        IDL
0105  45        LDA   R5
0106  A3        PLO   R3
0107  98        GHI   R8
0108  56        STR   R6
0109  D4        SEP   R4
010A  F811      LDI   #11
010C  BC        PHI   RC
010D  F895      LDI   #95
010F  AC        PLO   RC
0110  22        DEC   R2
0111  DC        SEP   RC
0112  12        INC   R2
0113  56        STR   R6
0114  D4        SEP   R4
0115  06        LDN   R6
0116  B8        PHI   R8
0117  D4        SEP   R4
0118  06        LDN   R6
0119  A8        PLO   R8
011A  D4        SEP   R4
011B  64        OUT    4
011C  0A        LDN   RA
011D  01        LDN   R1
011E  E6        SEX   R6
011F  8A        GLO   RA
0120  F4        ADD
0121  AA        PLO   RA
0122  3B28      BNF   #28
0124  9A        GHI   RA
0125  FC01      ADI   #01
0127  BA        PHI   RA
0128  D4        SEP   R4
0129  F811      LDI   #11
012B  BA        PHI   RA
012C  06        LDN   R6
012D  FA0F      ANI   #0F
012F  AA        PLO   RA
0130  0A        LDN   RA
0131  AA        PLO   RA
0132  D4        SEP   R4
0133  E6        SEX   R6
0134  06        LDN   R6
0135  BF        PHI   RF
0136  93        GHI   R3
0137  BE        PHI   RE
0138  F81B      LDI   #1B
013A  AE        PLO   RE
013B  2A        DEC   RA
013C  1A        INC   RA
013D  F800      LDI   #00
013F  5A        STR   RA
0140  0E        LDN   RE
0141  F5        SD
0142  3B4B      BNF   #4B
0144  56        STR   R6
0145  0A        LDN   RA
0146  FC01      ADI   #01
0148  5A        STR   RA
0149  3040      BR    #40
014B  4E        LDA   RE
014C  F6        SHR
014D  3B3C      BNF   #3C
014F  9F        GHI   RF
0150  56        STR   R6
0151  2A        DEC   RA
0152  2A        DEC   RA
0153  D4        SEP   R4
0154  00        IDL
0155  22        DEC   R2
0156  86        GLO   R6
0157  52        STR   R2
0158  F8F0      LDI   #F0
015A  A7        PLO   R7
015B  07        LDN   R7
015C  5A        STR   RA
015D  87        GLO   R7
015E  F3        XOR
015F  17        INC   R7
0160  1A        INC   RA
0161  3A5B      BNZ   #5B
0163  12        INC   R2
0164  D4        SEP   R4
0165  22        DEC   R2
0166  86        GLO   R6
0167  52        STR   R2
0168  F8F0      LDI   #F0
016A  A7        PLO   R7
016B  0A        LDN   RA
016C  57        STR   R7
016D  87        GLO   R7
016E  F3        XOR
016F  17        INC   R7
0170  1A        INC   RA
0171  3A6B      BNZ   #6B
0173  12        INC   R2
0174  D4        SEP   R4
0175  15        INC   R5
0176  85        GLO   R5
0177  22        DEC   R2
0178  73        STXD
0179  95        GHI   R5
017A  52        STR   R2
017B  25        DEC   R5
017C  45        LDA   R5
017D  A5        PLO   R5
017E  86        GLO   R6
017F  FA0F      ANI   #0F
```

```
0181  B5      PHI  R5
0182  D4      SEP  R4
0183  45      LDA  R5
0184  E6      SEX  R6
0185  F3      XOR
0186  3A82    BNZ  #82
0188  15      INC  R5
0189  15      INC  R5
018A  D4      SEP  R4
018B  45      LDA  R5
018C  E6      SEX  R6
018D  F3      XOR
018E  3A88    BNZ  #88
0190  D4      SEP  R4
0191  45      LDA  R5
0192  07      LDN  R7
0193  308C    BR   #8C
0195  45      LDA  R5
0196  07      LDN  R7
0197  3084    BR   #84
0199  E6      SEX  R6
019A  67      OUT   7
019B  26      DEC  R6
019C  45      LDA  R5
019D  A3      PLO  R3
019E  3688    B3   #88
01A0  D4      SEP  R4
01A1  3E88    BN3  #88
01A3  D4      SEP  R4
01A4  F8F0    LDI  #F0
01A6  A7      PLO  R7
01A7  E7      SEX  R7
01A8  45      LDA  R5
01A9  F4      ADD
01AA  A5      PLO  R5
01AB  86      GLO  R6
01AC  FA0F    ANI  #0F
01AE  3BB2    BNF  #B2
01B0  FC01    ADI  #01
01B2  B5      PHI  R5
01B3  D4      SEP  R4
01B4  45      LDA  R5
01B5  56      STR  R6
01B6  D4      SEP  R4
01B7  45      LDA  R5
01B8  E6      SEX  R6
01B9  F4      ADD
01BA  56      STR  R6
01BB  D4      SEP  R4
01BC  45      LDA  R5
01BD  FA0F    ANI  #0F
01BF  3AC4    BNZ  #C4

01C1  07      LDN  R7
01C2  56      STR  R6
01C3  D4      SEP  R4
01C4  AF      PLO  RF
01C5  22      DEC  R2
01C6  F8D3    LDI  #D3
01C8  73      STXD
01C9  8F      GLO  RF
01CA  F9F0    ORI  #F0
01CC  52      STR  R2
01CD  E6      SEX  R6
01CE  07      LDN  R7
01CF  D2      SEP  R2
01D0  56      STR  R6
01D1  F8FF    LDI  #FF
01D3  A6      PLO  R6
01D4  F800    LDI  #00
01D6  7E      SHLC
01D7  56      STR  R6
01D8  D4      SEP  R4
01D9  19      INC  R9
01DA  89      GLO  R9
01DB  AE      PLO  RE
01DC  93      GHI  R3
01DD  BE      PHI  RE
01DE  99      GHI  R9
01DF  EE      SEX  RE
01E0  F4      ADD
01E1  56      STR  R6
01E2  76      SHRC
01E3  E6      SEX  R6
01E4  F4      ADD
01E5  B9      PHI  R9
01E6  56      STR  R6
01E7  45      LDA  R5
01E8  F2      AND
01E9  56      STR  R6
01EA  D4      SEP  R4
01EB  45      LDA  R5
01EC  AA      PLO  RA
01ED  86      GLO  R6
01EE  FA0F    ANI  #0F
01F0  BA      PHI  RA
01F1  D4      SEP  R4
01F2  00      IDL
01F3  00      IDL
01F4  00      IDL
01F5  00      IDL
01F6  00      IDL
01F7  00      IDL
01F8  00      IDL
01F9  00      IDL
01FA  00      IDL
01FB  00      IDL
01FC  00      IDL
01FD  E0      SEX  R0
01FE  00      IDL
01FF  4B      LDA  RB
```

MACHINE CODE

P. V. Piescik, 157 Charter Rd., Wethersfield, CT 06109

We were talking about subroutines and how to call them and return from them to our calling routines. We looked at branching to/from subs, but this restricted us to having the caller on the same page of memory and returned to a single memory location every time. We'd prefer more flexibility, so this method was investigated and rejected. We also looked at SEP Technique, which allows us to return to a single register used as the caller's PC. It needed a little set-up, and was more flexible than branching, but "nesting" subroutines could be a problem at times. MARK Technique allowed us to return to any register, the caller's PC, by saving the caller's X and P on the stack. The set-up is about the same as for SEP Technique, but we also found that we could still run out of free registers to use for the caller- and sub-PC's...

...enter SCRT (Standard Call and Return Technique). I'm neither the first, nor the only, one to feel a shortage of registers! SCRT re-uses the same registers regardless of the nesting level, and is limited only by memory space available for the stack. A total of five registers are "dedicated" to SCRT, to be used only for SCRT purposes (of course, if you know that SCRT will not be used within some part of a program, these registers can be saved, used otherwise, and restored for more SCRT use). Again, R2 is the stack pointer and usually, RX. Every call/return will set X=2, so if the subroutine changes X, it may not have to restore it. R3 is always the PC, regardless of the nesting level! R6 is a "link" register which points back to the calling routine (one level up) at the location where R3 would be left if this were MARK or SEP technique. We can communicate between the caller and subroutine via this link. R6 also contains the return address which will be reloaded into R3 upon return.

The PC and Link are maintained in an orderly fashion by two SCRT Routines --CALL and RETURN, with R4 and R5 dedicated as their respective PC's. CALL and RETURN are invoked using SEP Technique; however CALL "returns" to the subroutine being called, rather than CALL's caller. The subroutine "returns" to RETURN, which finally returns to the original caller. This sounds chaotic, but it might be easier to think of CALL as a logical prefix and RETURN as a logical postfix to each subroutine. Since they are physically displaced from the code of the subroutines in memory, we might also think of them as separate, intermediate steps. In any case, once we see what they do, they are (thankfully) transparent to their users.

Since R6, the link, points to where R3 did in the caller, CALL must copy R3 to R6; and, since CALL doesn't know (or care!) if this is the first, 2d, or "n-th" level of nesting, the previous link must be saved on the stack. Finally, R3 will be loaded with the address of the subroutine, and SEP-ed as the PC again, only this time, for the sub. The entry address is supplied by the caller as "in-line data" (so called because it is in line with the instructions of the routine); once R6 points to where R3 did in the caller, it may be used to fetch this address.

RETURN has one less thing to do than CALL did, since we don't need to save the sub-PC (R3); unlike the other techniques, SCRT will re-load the sub-PC on the next call, and we can't re-prime R3 at the end of the sub, since it will be used as the PC of other routines before we get back to any particular subroutine. Undoing the rest of the call, R6 will be copied back to R3 to restore the caller's PC. R6 will then

be restored by popping the previous link-value from the stack. Lastly, R3 will be SEP-ed into action as the caller's PC.

To keep the code in ascending order, I'll show the expanded initialization which now sets up R2 as the stack pointer, R3 as the main PC, R4 as the CALL PC, and R5 as the RETURN PC:

```
0000  F8 00 B2          LDI 0;PHI 2           ..this page
0003  B4 B5             PHI 4;PHI 5           ..also this page
0005  F8 FF A2          LDI #FF;PLO 2         ..(R2)=00FF
0008  E2                SEX 2                 ..stack ready
0009  F8 16 A4          LDI A.0(CALL);PLO 4   ..CALL PC ready
000C  F8 28 A5          LDI A.0(RETURN);PLO 5..RETURN PC ready
000F  F8 00 B3          LDI A.1(MAIN);PHI 3   ..
0012  F8 36 A3          LDI A.0(MAIN);PLO 3   ..MAIN PC ready
0015  D3                SEP 3                 ..go, MAIN, go!
```

This set-up, including CALL and RETURN shortly, is good for almost all the 1802 programming you'll ever do! Anticipating that, I saved 4 bytes by loading R4.1 and R5.1 right after R2.1, while D still contained #00. I didn't load R3.1, though, since MAIN may not always be on page #00, especially if you adopt my programming habits.

```
0015  D3                SEP 3                 ..gosub
0016            CALL:   ORG *
0016  E2                SEX 2                 ..insure stack ptr
0017  86 73             GLO 6;STXD            ..save R6 link
0019  96 73             GHI 6;STXD
001B  83 A6             GLO 3;PLO 6           ..copy R3 to R6
001D  93 B6             GHI 3;PHI 6
001F  46 B3             LDA 6;PHI 3           ..load R3 with addr
0021  46 A3             LDA 6;PLO 3           .. and advance link
0023  30 15             BR CALL-1             ..reset R4 and gosub

0027  D3                SEP 3                 ..return to caller
0028            RETURN: ORG *
0028  86 A3             GLO 6;PLO 3           ..restore R3 from R6
002A  96 B3             GHI 3;PHI 3
002C  E2 60             SEX 2;IRX             ..insure stack ptr and
                        ..                      prime POP
002E  72 B6             LDXA;PHI 6            ..restore R6 from stack
0030  F0 A6             LDX;PLO 6
0032  30 27             BR RETURN-1           ..reset R5 and return
```

Note that R(X) is not left to chance in either case, since a routine may have changed X. At 001F-0022, we load R3 with the address of the sub, which is 2 bytes of inline data, and we use LDA 6 (46) instructions for this. If the subroutine also needs some data which is stored inline, it will also use 46 instructions to get each byte of data. There MUST BE as many 46's as there are bytes of data, or the return will be incorrect! Advancing R6 now is the same as advancing R3 in the caller, since R3 will be restored from R6; too few 46's and the return is into unused data, or too many 46's and the return is beyond the next instruction.

I also saved a byte by getting double-duty from the D3 at 0015! It belongs to CALL, but is also used to SEP into MAIN initially.

One problem with SCRT is that it uses the accumulator to manipulate R3 and R6, so D can't be used to pass a byte of data to/from a subroutine. If we try to save D on the stack, it will be buried under R6. The SCRT routines have been enhanced to preserve D at the expense of RF.1:

```
0015  D3                SEP 3
0016            CALL:   ORG *
0016  BF                PHI F                  ..copy D to RF.1
0017  E2                SEX 2
0018  86 73              GLO 6;STXD
001A  96 73              GHI 6;STXD
001C  83 A6              GLO 3;PLO 6
001E  93 B6              GHI 3;PHI 6
0020  46 B3              LDA 6;PHI 3
0022  46 A3              LDA 6;PLO 3
0024  9F                 GHI F                  ..restore D from RF.1
0025  30 15              BR CALL - 1

0027  D3                SEP 3
0028            RETURN: ORG *
0028  BF                PHI F                  ..copy D to RF.1
0029  86 A3              GLO 6;PLO 3
002B  96 B3              GHI 6;PHI 3
002D  E2 60              SEX 2;IRX
002F  72 B6              LDXA;PHI 6
0031  F0 A6              LDX;PLO 6
0033  9F                 GHI F                  ..restore D from RF.1
0034  30 27              BR RETURN - 1

0036            MAIN:   ORG *                  ..next available byte
```

OK, now what happens to the infamous ADD routine when we use SCRT? Do we need additional set-up as with SEP and MARK techniques? Etc.? From the original version of long ago, ADD grew by 1 byte for SEP and by 3 bytes for MARK; now, we can actually shrink it by 1 byte! We no longer have to wrap it back to the top to reset its PC, since SCRT will handle that for subsequent calls, and the return at 0095 is only a 1-byte D5 instruction. No additional set-up is required specifically for ADD; the new initialization is good for ADD and any other subs we have. The call to ADD is 3 bytes:

```
0036  D4                SEP 4                  ..invoke CALL
0037  0080              ,A(ADD)                ..entry address
```

ADD starts at 0080 again, and we can move the data up one byte since the return is shorter:

```
0080            ADD:    ORG *
0080  8E 73 9E 73        GLO E;STXD;GHI E;STXD..save RE
0084  F8 00 BE           LDI A.1(ONE);PHI E
0087  F8 96 AE           LDI A.0(ONE);PLO E     ..RE=A(ONE)
008A  EE                 SEX E                  ..X=E
008B  4E F4 1E 5E        LDA E;ADD;INC E;STR E..(SUM)=(ONE)+(TWO)
008F  E2 60              SEX 2;IRX              ..RX on stack, prime POP
0091  72 BE F0 AE        LDXA;PHI E;LDX;PLO E ..RE restored
0095  D5                 SEP 5                  ..return via SCRT
0096  01          ONE:    ,1
0097  02          TWO:    ,2
0098  xx          SUM:    ORG *
```

One advantage of SCRT, mentioned already, is the ease of passing inline data. Note that you're not concerned with the address of that data; R6 points to it "automatically"! Another advantage is that once you know

which register is your PC, you can use that register to provide some or all of the local (within the subroutine) addresses, even if the sub is not on the page for which it was originally written:

```
0084  93 BE            GHI 3;PHI E        ..RE on this page in
                  ..                         any case
0086  83               GLO 3              ..D=this address (lo
0087  FC 0F AE         ADI 15;PLO E       .. byte) + 1, then add
                  ..                         distance to ONE to
                  ..                         load RE.0
```

Getting the page address into RE.1 is obvious, and hopefully, we won't end up crossing a page boundary between this point and the location of the data. At #0086, we'll get our current address, plus 1, into the accumulator; (D)=87. For those who think this should be 86, let's see what happened. P=3 and (R3)=0086 as we come to the 83 instruction. On every instruction fetch, R(P) is incremented as the 4-bit I and N registers are loaded; all this occurs in the first machine cycle (S0). In the second machine cycle, the instruction is executed (I=8, N=3) and R3.0 is copied into D, AFTER having been incremented to 0087 in the first cycle. Anytime we get our location this way, D will contain the byte-address of the next instruction. It turns out that we're not saving any memory with this method, so ONE is still at 0096, and #96-#87=#0F; we add #0F to where we are, drop it into RE.0, and RE points to our data. If we want to include this ADD routine in some other program, we won't have to worry about where the data is, or to remember to make the change in the code! The data will always be +0F from this instruction, and no changes are needed unless we cross a page boundary between the 93 instruction and the data. One less thing to do is one less chance to build in a bug!

Since we always know which register is PC, we can also stop any routine for debugging purposes with only a 1-byte change instead of two! P=3 in all cases (except in CALL and RETURN), so instead of our old "BR *" which held us in one place, we can use a DEC 3 (#23) instruction. Not only is this half the trouble of changing 2 bytes, but it may also save the problem of having only 1 byte to spare in cases where the second byte we'd change belongs to code we need. Like the 83 at 0086 above, during the S0 cycle R3 is incremented when the 23 instruction is fetched; during the S1 cycle R3 is decremented and ends up where it started!

PROBLEM ANALYSIS and PROGRAM DESIGN. You probably picture "programming" as sitting at a terminal, keypunch, etc. and banging in lines of instructions in some language, right? WRONG. If it's done right, that's only the last 10-20% of the job, including testing! The major part is the analysis of the problem and the design of the program. Cutting short these phases will lead to your picture of long hours at the terminal, spending 50% or more of your time testing and debugging, which is very frustrating. In Systems Programming, John J. Donovan lists 6 steps for software design: 1) specify the problem; 2) specify the data structures; 3) define the format of the data structures; 4) specify the algorithm (procedural steps); 5) look for modularity (i.e., capability of one program to be subdivided into independent programming units); 6) repeat 1 through 5 on modules.

Donovan, John J.: Systems Programming, McGraw-Hill Book Co., 1972.

I didn't do this explicitly for ADD, but all the details came out in the discussion. (We jumped into the middle, where it "looked like computing," since starting at the beginning is less interesting, unfortunately.) Let's back up and do it right, before we develop bad, sloppy habits.

I understated the problem: add two numbers; that left a lot unsaid. If the problem had been a little more complex, we'd have had to make too many assumptions about the details, and that leads to trouble. By the numbers: 1) Problem: add two numbers found in memory and leave the result in memory; 2) Data Structures: two addends, one sum; 3) Format: all are 8-bit, unsigned integers; 4) Algorithm: addend.one + addend.two = sum; 5) Modularity: none (this is primitive); 6) not applicable without modularity.

(1) now tells where the numbers are; before they might have been in registers, or terminal input. (2) and (3) tell how many numbers there are altogether (we might have replaced one addend in memory with the result) and what they look like, so we can tell how much memory they need.

(4) is still rather simple, and we can and should add details until we're sure we understand the process--4.1) free a register for use as a pointer by pushing its contents on the stack; 4.2) initialize the pointer to contain the address of addend.one; 4.3) make the pointer the index (RX) register; 4.4) addend.one + addend.two = sum; 4.5) restore the stack pointer (R2) as RX; 4.6) restore the contents of the register; 4.7) return. (4.4) still looks like the old (4), and we may need to keep going: 4.4.1) load the accumulator (with addend.one) via the pointer register and advance the pointer; 4.4.2) add; 4.4.3) advance the pointer to the 3d location (of sum); 4.4.4) store the result (from the accumulator) via the pointer.

The way we advanced the pointer in (4.4.1) and (4.4.3) implies that the numbers occupy sequential memory locations, and the sum is in the last location. This probably should have been specified in (2), or some other arrangement, like having registers pointing to the numbers provided by the caller, might be specified in (1). If we're handling all the data from start to finish, we'll establish the location and arrangement of the data once we've determined all the structures (2) and the formats (3). The remaining details of the algorithm are now almost to the point where actually 1802 instructions would be next; if the problem is still a mystery, you have little choice but to go over it again.

Looking for modularity (5) is just a matter of chopping the problem into chunks you can handle, without having to deal with all the details of a big job at once. Too many details create confusion, and confusion creates errors. A good rule of thumb is that the function of a module can be told in one sentence. If you can't do that, chop it smaller. In most cases, a module should require no more than about 188 bytes (3/4 page) to reduce the chances of errors, which improves reliability. If we carry modularization far enough, we'll arrive at the structured programming ideal--functional uniqueness.

The whole point to these steps is that if YOU don't understand what you are trying to do, you'll never get your idiot computer to do it!

(2) and (3) describe the data which is part of the problem (addends and sum for ADD), and may include auxiliary data which is not part of the problem, but part of the computer solution. We could include the address of the numbers for ADD, or at least the first one the way we did it; I've omitted it because the address is the immediate data in the instructions.

Data basically comes in three types, excluding floating-point and (at least on IBM machines) packed decimal: character, numerical, and logical or Boolean. All I/O is done with character data, in ASCII code (except if you have equipment using other codes) and unless the data is a number on which we'll be doing arithmetic, we'll leave it in ASCII in memory. Each character occupies 1 byte, with the low-order 7 bits containing the ASCII code; the highest-order bit is parity, which we'll usually strip off (set to 0) after input, and it's available for special use. If the I/O device uses parity, this bit will be a 1 to bring the total number of 1-bits in the byte to an even (even parity) or odd (odd parity) total, and we might check this on input to insure the validity of the data. We don't need parity in memory, so we'll get rid of it to leave this bit for things like indicating that a byte is the last of a string, instead of using another whole byte to contain some special code (delimiter) to indicate the end. Usually, we'll need more than one character for a data item ("157 Charter Rd.,") and the item's length will be the number of characters (16; the " are delimiters, not part of the string). In this case, "157" is a number, but we won't be adding/subtracting/etc. so it's not numerical data!

If it were numerical data, doing arithmetic in ASCII is awkward and inefficient, so we'd convert "157" (#313537) to binary (#9D) and save 2 bytes in the process. All the bits in the byte are now part of the numerical value, unlike the high-order 3's in the ASCII. Obviously, while 157 fits in 1 byte, we may be dealing with different ranges of values for data, and we'll require different numbers of bits to hold them. Your 40-hour work-week fits in 6 bits (101000), and your $20K salary fits in 15 (100111000100000) if it's in whole dollars, or in 18 bits (111101000010010000000) if it's in pennies because we're working with integers. That's 1-3 bytes, and the amount of memory needed for each item will depend upon the range of values it can have.

There are conventions we should examine before going wild with oddball lengths for numerical data. Common lengths are 1, 2, 4, and 8 bytes (8-64 bits) or longer, but always twice as long as the next shorter; if we multiply two 8-bit values, the product will have up to 16 bits (the length of the product is the sum of the lengths of the multiplicand and multiplier). Obviously, we can't go shorter than 8 bits with byte addressing. Hey, what about negative numbers? We might have negative results, for example, if we subtract 1 from 0. In hex, #00 - #01 = #FF; and, #FE + #01 = #FF. If we see the work, we'll know that the first answer is -1 and the second answer is 255, but if we're just handed the value, which is it?

That depends on whether we're talking in signed or unsigned numbers, and that is sometimes an arbitrary decision. Addresses are unsigned numbers, and we may decide that all our values are unsigned if the problem permits. If not, the MSB (most-significant bit) is the sign, which is 0 for + and 1 for -; zero is still #00, so it's considered to be positive! The sign is not part of the value, so signed and unsigned integers with the same number of bits (including the sign) have different ranges of values.

The number of values, however, is the same either way. For an 8-bit integer, there will always be 256 (2^8) values; 0-255 (0 to 2^8-1) if it's unsigned; -128 to +127 (-($2^{(8-1)}$) to $2^{(8-1)}$-1) if it's signed. When we subtracted 1 from 0, we "wrapped around" our 8-bit value, just as you can wrap around the counter on your cassette recorder, or a cheap adding machine would show 99,999.99 after the same subtraction. It may be a little easier to look at the adding machine first, then carry our findings to the computer.

Most adding machines will show one more digit in the total than you can enter on the keys, for overflow, so you don't lose significance too easily. We could enter up to 9,999.99 on the example machine and the extra 9, in the "sign" position, clues us to overflow when adding, or that we've gone negative in subtracting. If we've subtracted and the extra 9 shows up, the result is the 10's complement of the true value. The 10's complement of a number is the value which brings the number to 10, or to 0 if we run out of places to carry into; in this case, it's obviously 1. The 10's comp. is also 1 more than the 9's comp., which would bring each column to a total of 9; in this case, 00,000.00. If you've ever used such an adding machine, you also know that this odd-looking number only shows up if you print it on the tape (I/O), and that if you continue to add and subtract to a positive total, it prints out correctly.

The same thing happens in binary, except that our digits are bits and the complements are 2's and 1's; the sign-bit is that extra, overflow digit. Taking 0000 0001 from 0000 0000 gave us 1111 1111, so the extra digit has shown up, clueing us to a negative value. Since binary can get messy with a lot of carrying, let's find the 1's complement by bringing each column to 1: 0000 0000 (just change all 1's to 0's and all 0's to 1's). Adding 1, we get 0000 0001, and finally, we change the sign to get -1 (without the leading 0's). As with the adding machine, going negative somewhere during a chain of addition and subtraction will not affect the total, and negative numbers are a "problem" only for I/O, multiplication and division. In these cases, we look for a "-" character on input, convert the number as if it were positive, then subtract it from 0 (which is easier than getting the 1's comp. and adding 1); for output we'll look for the sign-bit = 1, then subtract the number from zero, convert it to ASCII, and create a "-" to output to the left or right of it. For multiplication and division, we'll use only the absolute values of numbers, keeping the signs separate; if the algebraic sign of the result is -, we'll negate the result.

If we're really on the ball, we'll do the sign algebra for all operations, since an unexpected change of sign (e.g., a negative result from adding two positive integers) indicates "overflow" and a loss of significance.

Data which is neither character nor numerical is logical, or Boolean, and has two possible values: true and false. Only 1 bit is required to store a Boolean value, so 8 conditions may be represented by a single byte if we want to do it that way. At other times we may want to make a logical decision based on data which isn't Boolean and will be formatted according to other requirements. A 0-bit is false, and a 0-value for other data types is also false; a 1-bit is true, and any non-zero value for other types of data is also true. For example, if we get a 1-byte error code back from our subroutines, where #00 is OK and 01-FF indicate the type of error, we'd ask, "Error?" by checking for 0, or Error=False in English. If that's the case, we'd proceed normally. If

the byte is non-zero (Error=True in English), we'll output the number to the user, and probably stop to wait for his correction. Or, we'd translate the number into a text message instead of giving the poor user a mere code-number. In a more sophisticated environment, we might use the error code to determine our action: send a message and keep going, send a message and stop for the solution, send a message and abort, or even, attempt an automatic recovery (like re-reading a floppy 10 times) before yelling for help.

1861 and other video or hardcopy graphics involve bit patterns which are either picture elements (pixels) or color codes, etc., and these are closer to logical data than anything else. We might not think in terms of "true" and "false" (unless the question is "lighted?"), but the data makes even less sense as a character or number.

When we've determined, possibly after several repetitions of Steps 1-5, what all the data is, we'll allocate memory for it. We might not have a particular location in mind, yet, but we'll establish the order of the items so we'll have relative addresses from the start of the data; when we know that address, it's simple addition to convert the addresses to absolute. I like to put all the constants first (titles, text messages, etc.), especially if the program may be in ROM eventually. The variables, which will have to be in RAM, will all be together, and you won't have to rearrange things as much to split the ROM-ables from the RAM-ables; putting the variables on a separate page of memory helps even more. I also put fixed-length variables at lower addresses than variable-length variables, especially if the maximum length is limited only by the amount of memory on the machine. For example, I have a program which handles text as one long line for input and editing, and the only limit on the length of this line is having memory to hold it. If I add memory to the machine, both the end of memory and the max. end of the line move outward together by changing only 1 parameter in the program. (For output, a variable establishes the length of a line on the printer, video screen, etc., so the program determines where to break the text into "lines". The untouched text can be output on devices with different line-lengths.)

Usually, all the data will fit on one page, with the exception of items like video-screen-memory and expandable text. For the data page, one register becomes a "base pointer," and any item can be reached by putting its low-order address into that half of the register; several items which are needed in a messy order of access may need additional, temporary base registers, created by copying the high byte of the original base into more registers. If there's a separate video buffer and/or text, each will have it's own base register, either permanently or temporarily (why tie up a register all the time for occasional use?).

Next time: "top-down" design and "bottom-up" coding.

Hi Ray: I just received my issues of the VIPER. I think it's great. I've read through them and have not seen anything on medium res. 64 by 64. I wrote my own version of CHIP-8--I call it CHIP-BETA. It has all the CHIP-8 instructions plus the I/O instructions.from 8X to run the Simple Sound and player 2 Hex keypads. The main feature is 64 by 64 graphics. I couldn't live without it! The DRAW program from the 311 manual can be used immediately without conversion to draw things in medium res. I plan to sell copies of CHIP-BETA for $5.00 a copy. If there is anybody interested, write me.

Also, in the Sept. 81 issue of BYTE magazine, there is a circuit called the "Sweet Talker" It runs through a parallel port. I believe it could be interfaced to the VIP using the Q-line and operated like the Simple Sound board. This circuit would give a 4K VIP a 1000 plus word vocabulary. Has anybody done it yet?

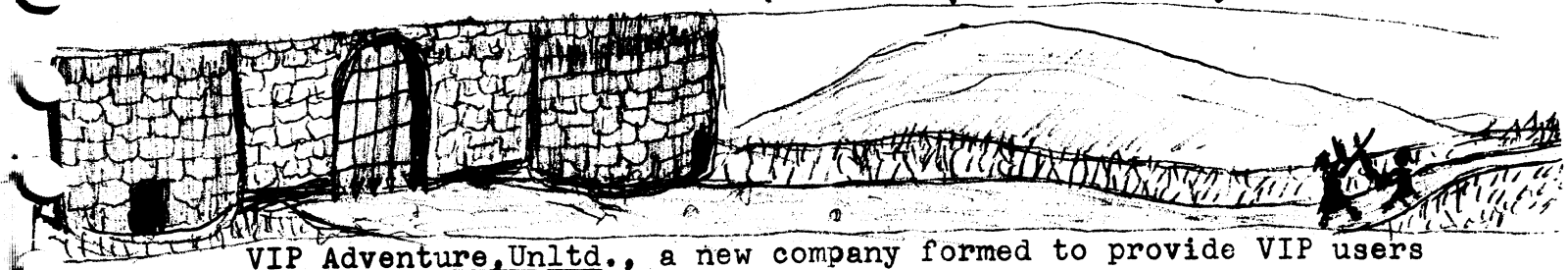
FSS Softwarw company puts out a version of FORTH for the Super-ELF. Does anybody have it running on a VIP? I think FORTH would be a great third or forth (pun intended?RS) language for the VIP. also, is it possible and has anyone built a five to eight slot I/O board that is programmable for the VIP? This would be great for using a printer, modem, sound, voice, etc.

I tried writing "Asteroids" but couldn't do the asteroid part very well. Is there anybody out there with a copy I could buy or trade for? Well, congratulations on the VIPER, and keep up the excellent work. I plan on documenting my VIP double array function and sending it in. I use it in a program to computer 16 by 16 mazes. Well, best of luck in the future. Good computing!!

Ron Applebach 17106 E. Bethany Circle Aurora, CO 80013

ADVENTURE

FOR YOUR VIP!



VIP Adventure,Unltd., a new company formed to provide VIP users with the enjoyment of adventure programs is proud to announce the release of our first program entitled Quest of the Enchanted Sword. It takes place in medieval times in the legendary kingdom of Camelot, several years after King Arthur's death.It will run on a 4K VIP with an ASCII keyboard.In addition, there are three "mini adventures" on the flip side of the cassette that will run on a 4K VIP with Tiny Basic. The cost of the cassette is $8.95,shipping charges included. Make checks payable to Nathan Gopen, and mail orders to:

VIP ADVENTURE, UNLTD.
168 Pond St. Sharon, MA.
02067

Many more adventures soon to come! 3.05.24

COSMAC VIP Autocall System

by George S. Gadbois, W3FEY

Telephone Companies provide Speedcall(R) service for about $5.00 per month. You can have your own autocall system using a microprocessor for a modest capital investment and almost no operating cost. A simple autocall system using the RCA COSMAC VIP is described here. If you don't like the features, just change the software. Try getting Ma Bell to change her service features.

The RCA 1802 microprocessor is an excellent choise for control functions around the home or radio shack. Building a microprocessor system from the ground up is an interesting and sometimes frustrating exercise. There are many moldering microprocessor repeater control* projects around to attest to the frustrating nature of building supposedly simple microprocessors. If you enjoy such exercises, fine, but if you just want to get on with the job, let someone else do the dog work for you. The RCA VIP has all the hardware you need for simple I/O projects without burdening your pocketbook. The buffered byte I/O interface is easily interfaced with external logic or relays=

The VIP Autocall system described here requires a 4K RAM to support the high resolution graphics display used for operator prompts and data display. If you'll settle for less sophistication in the graphics, you can use a 2K RAM. One page, 256 bytes, is set aside for storage of phone numbers. This permits twenty-one 12-digit numbers to be stored. If you want to store more numbers, you can reduce the maximum length or increase the size of the buffer area. Two additional pages of RAM can easily be made available in a 4K VIP.

The VIP Autocall uses a Mostek MK5089N integrated tone dialer to generate the standard Touchtone(R) frequencies. If you are limited to a dial pulse exchange, you can change the program to generate dial pulses. The circuit chown in Fig. 1 will generate the full 4 by 4 tone matrix, opening remote control possibilities not available with a standard tone pad. As written, the "#" key is used as the "end of number" key; however, this can be easily changed in the software.

A modified CHIP-8X interpreter with a high resolution display routine is used in this system. This choice was made because of the CHIP-8X output to port instruction and the pleasing appearance of the high resolution graphics. CHIP-8 could be used with a machine language subroutine for the tone output. Note that the normal CHIP-8X byte <u>input</u> instruction got displaced by the high resolution display routine. But is is not needed for this application.

The modified CHIP-8X interpreter occupies memory locations 0000 to 02FF and is shown as a Hex dump in Table I. The Autocall program is shown with comments in Table II. Because an excess of memory was available, the subroutines and main program were widely separated to make modification and debugging easy. A convenient method for operator message generation is provided by storing messages in ASCII code and then looking up the appropriate character display patterns in a memory buffer. The modified CHIP-8X and message generator were written by Steve Houk of RCA for a morse keyer program. The Autocall program uses the ASCII tables only for internal message generation so no ASCII keyboard is required to use the Autocall system. If you want a new message you can enter the ASCII equivalent Hex codes. The message bufferis on page 0600-06FF. NUL code 00 denotes end of message.

0000	91BB	FF04	B2B6	F8CF	A2F8	02B1	F806	A190
0010	B4F8	1BA4	F801	B5F8	FAA5	D496	B7E2	94BC
0020	45AF	F6F6	F6F6	3244	F950	AC8F	FA0F	F9F0
0030	A605	F6F6	F6F6	F9F0	A74C	B38C	FC0F	AC0C
0040	A3D3	301B	8FFA	0FB3	4530	4022	6912	D400
0050	0001	0101	0101	0101	0101	0100	0102	0101
0060	007C	7583	8B95	B4B7	BC91	EB70	D92F	9905
0070	30A0	45FA	0FAF	329E	46FA	3FF6	F6F6	2252
0080	E206	FA7F	FEFE	3B8A	1D1D	FE3B	8E1D	F1AC
0090	008F	BC12	075C	8CFC	08AC	2F8F	3A94	D4D4
00A0	F8F0	AD30	7256	168F	5616	308E	00EC	F8D0
00B0	A693	A78D	32D9	06F2	2D32	BEF8	01A7	46F3
00C0	5C02	FB07	32D2	1C06	F232	CEF8	01A7	06F3
00D0	5C2C	168C	FC08	AC30	E6F8	1C2F	9F3A	EAD4
00E0	F804	BFF8	FFAF	9BBC	94AC	945C	30DA	42B5
00F0	42A5	D48D	A787	32AC	2A27	30F5	3AB3	30D9

0100	0000	0000	0047	A398	56D4	F881	BCF8	95AC
0110	22DC	1256	D406	B8D4	06A8	D464	0A01	E68A
0120	F4AA	3B28	9AFC	01BA	D4F8	81BA	06FA	0FAA
0130	0AAA	D4E6	06BF	93BE	F81B	AE2A	1AF8	005A
0140	0EF5	3B4B	560A	FC01	5A30	404E	F63C	3C9F
0150	562A	2AD4	0022	8652	F8F0	A707	5A87	F317
0160	1A3A	5B12	D422	8652	F8F0	A70A	5787	F317
0170	1A3A	6B12	D415	8522	7395	5225	45A3	06FA
0180	0FB5	D445	E6F3	3A82	1515	D445	E6F3	3A88
0190	D445	0730	8C45	0730	84E6	6226	45A3	3688
01A0	D43E	88D4	F8F0	A7E7	45F4	A586	FA0F	3BB2
01B0	FC01	B5D4	4556	D445	E6F4	56D4	45FA	0F3A
01C0	C407	56D4	AF22	F8D3	738F	F9F0	52E6	07D2
01D0	56F8	FFA6	F800	7E56	D419	89AE	93BE	99EE
01E0	F456	76E6	F4B9	5645	F256	D445	AA86	FA0F
01F0	BAD4	0000	00E6	63D4	30F5	0229	00E0	004B

0200	1300	017A	4270	2278	2252	C419	F800	A09B
0210	FA0C	B0E2	E2E2	3C14	3418	9832	21AB	2B8B
0220	B888	3203	7B28	3004	019B	FF03	BBD4	019B
0230	AD06	FA07	BE06	FA3F	F6F6	F622	5207	FA7F
0240	FEFE	3B46	1D1D	FE3B	4A1D	F1AC	8DBC	45FA
0250	0FAD	A7F8	D0A6	94AF	8732	AF27	4ABD	9EAE
0260	8E32	6C9D	F6BD	8F76	AF2E	3060	9D56	168F
0270	5616	3056	D3EC	F8D0	A694	A78D	32A7	06F2
0280	2D32	86F8	01A7	46F3	5C02	FB07	329A	1C06
0290	F232	96F8	01A7	06F3	5C2C	168C	FC08	AC9C
02A0	7C00	BCFB	103A	7BF8	FFA6	8756	12D4	018D
02B0	A787	3274	2A27	30B1	01D4	3CB5	80E2	20A0
02C0	34BC	9832	29AB	2B8B	B888	3273	7B28	3074
02D0	92BD	F89F	ADF8	C0BC	94AC	F8AA	5C94	BCAC
02E0	0CFB	9132	ECF8	915C	F801	5DD4	F800	5DD4
02F0	E265	22D4	0000	0000	0295	02D0	0281	004B

0400	E0E0	E0E0	E0E0	E0E0	E0E0	E0E0	E0E0	E0E0	ASCII PATTERN
0410	E0E0	E0E0	E0E0	E0E0	E0E0	E0E0	E0E0	E0E0	
0420	779A	B5BF	7CBA	C9A0	6EC4	CE90	9794	78B0	INDICES
0430	2E72	424D	3040	4432	4648	A89D	8BA4	8985	
0440	8100	6A18	6614	1621	0325	3406	540D	602E	
0450	511C	583C	2910	610C	095B	386E	ACC4	D3DC	
0460	1A1C	A0DE	CEFE	273F	1351	3E90	AC76	D0BB	
0470	040A	1818	7372	E7D8	08E5	806E	A8C4	A4B6	
0500	E0A0	E0A0	A0E0	A0A0	C0A0	A040	A0A0	E0E0	ASCII DISPLAY
0510	A0A0	A0A0	E080	E080	E080	8080	E0A0	A0E0	
0520	20E0	80A0	A0E0	4040	40E0	4040	4040	E0A0	PATTERNS
0530	A0A0	E020	2020	20A0	E020	4080	E080	4020	
0540	E080	E020	E080	E0A0	E0A0	E020	20E0	20E0	
0550	20E0	A0E0	8080	8080	E0A0	C0A0	A040	4040	
0560	E0A0	A0A0	A040	C0A0	A0A0	C0A0	E0A0	C080	
0570	8080	C040	4040	E000	0000	0000	40E0	40E0	
0580	4000	E0A0	80E0	20E0	0080	4020	4080	4020	
0590	0040	E040	0000	E000	0000	4040	4000	4000	
05A0	4040	0000	00E0	00E0	0040	0040	0080	4020	
05B0	0020	4080	00A0	A000	0000	A020	4080	A000	
05C0	A040	A000	6020	2020	6040	E020	E040	0040	
05D0	A040	0040	A000	4040	A0A0	A040	0000	0000	
05E0	0000	0000	00E0	1E77	A8FA	88D5	BE8F	2C5F	
05F0	C890	0984	DFFC	FFEF	30A8	911E	EF17	3312	
0600	5649	5020	4155	544F	4449	414C	4552	0031	MESSAGE BUFFER
0610	2D49	4E50	5554	2C32	2D53	454E	4400	494E	
0620	5055	5420	5080	4F4E	4520	2020	2020	4E55	
0630	4D42	4552	2E20	454E	4420	5749	5448	2022	
0640	4622	0053	454E	4420	4E55	4D42	4552	0044	
0650	4F4E	4500	4E55	4D42	4552	2053	544F	5245	
0660	4400	494E	4445	583D	0049	4E44	4558	2054	
0670	4F4F	204C	4152	4745	0020	2020	2020	2020	
0680	5459	5045	2049	4E20	4D45	5353	4147	4520	
0690	5448	454E	2050	5245	5353	2020	2020	2020	
06A0	5245	5455	524E	2046	4F52	204F	4E45	2020	
06B0	584D	4954	204F	5220	2020	2020	2020	2020	
06C0	4C49	4E45	2046	4545	4420	464F	5220	2020	
06D0	5245	5045	4154	2058	4D49	542E	0000	0000	
06E0	4255	4646	4552	2046	554C	4C21	008F	2C5F	
06F0	C890	0984	DFFC	FFEF	30A8	911E	EF17	3312	

VIP Autocall Table II

Address	Code	Comments
0300	6EFF	Clear output to MK5089
0302	FEF8	
0304	6900	Message index: "VIP Autodialer"
0306	2750	Display message
0308	6880	Delay
030A	27A0	
030C	690F	Message: "1-Input, 2-Send"
030E	2750	Display
0310	F60A	Wait for Hex key
0312	4601	Go if 01 (input)#)
0314	131C	to 131C
0316	4602	Go if 02 (send)#)
0318	1380	to 1380
031A	130C	Else loop back on invalid input to 130C
031C	2700	Go to buffer index subroutine
031E	691E	Message: "Input Phone number. End with 'F' "
0320	2758	Display
0322	1350	Go to 0350
0324	AA00	Phone number buffer index; points to 0A00
0326	FA1E	
0328	F00A	Wait for hex key
032A	F055	Store digit just entered
032C	400F	Go if 'F' (entry finished)
032E	1340	to 1340
0330	7A01	ELSE Increment index pointer
0332	7C01	Increment count
0334	F029	Display digit entered
0336	D455	at XY location of V4, V5
0338	7405	Increment X co-ordinate of display: X=X+5
033A	4C0C	Test for 12 digits in buffer
033C	1340	Yes? then buffer full; Go to 0340
033E	1324	No? Then go to 0324 and get another
0340	6954	Message: "Number stored"
0342	2758	Display
0344	6880	Delay
0346	27A0	
0348	130C	Go to start of program; menu at 030C
0350	6C00	Count = 00
0352	6400	Next line of display
0354	7510	
0356	1324	Go to 0324

Phone Number Send Routine

0380	6C00	Count = 00
0382	6943	Message: "send number"
0384	2750	Display
0386	2700	Go to Index number subroutine
0388	6803	V8=03; tone duration (about 50msec)
038A	6400	Next line of display; V4=00 (X co-ordinate)
038C	7510	V5=V5+10; (Y co-ordinate)
038E	AA00	Phone number buffer index; points to 0A00

address	Code	Comments
0390	FA1E	Index into buffer
0392	F065	V0=digit to be sent
0394	7C01	Increment count
0396	400F	Test for 'F' (end of digit string)
0398	13B6	Yes? then Go to 03B6
039A	7A01	No, then increment buffer pointer and
039C	F029	Display digit
039E	D455	
03A0	7405	Increment X co-ordinate
03A2	A900	Point to Touchtone (TM) code
03A4	F01E	
03A6	F065	V0= 2 of 8 code for output to MK5089
03A8	F0F8	Output V0 to I/O port
03AA	27A0	Go to delay Subroutine
03AC	FEF8	Clear output
03AE	27A0	Go to delay subroutine
03B0	4C0C	Test for 12 digits sent; VC=count
03B2	13B6	Yes? Then Go to 03B6
03B4	138E	No? Then to 038E and get another digit
03B6	694F	Message: "Done"
03B8	2758	Display the message
03BA	6880	Delay
03BC	27A0	Subroutine
03BE	1380	Go to next phone number

Phone Number Buffer Index Subroutine

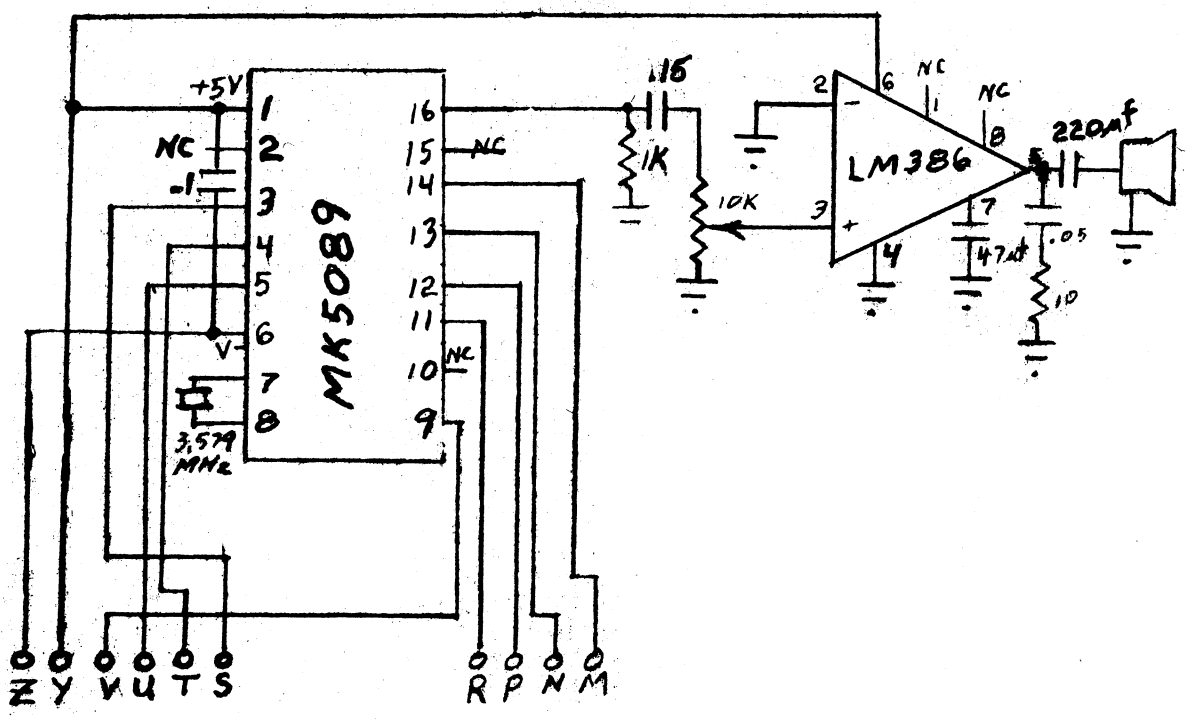
0700	6962	Message:"Index="
0702	2758	Display message
0704	F60A	Wait for Key Press
0706	F629	Display MSD of index number
0708	7405	V4=V4+05; Increment X co-ordinate
070A	D455	Display it
070C	4600	If V6=00 Then
070E	1716	Go to 0717, else
0710	4601	If V6=01 THEN
0712	1728	Go to 0728, else
0714	173E	Go to Error message
0716	F60A	Wait for Key Press
0718	F629	Display LSD of index number
071A	7405	V4=V4+05; Increment X co-ordinate
071C	D455	Display it
071E	8A6E	VA=2*V6
0720	8A64	VA=VA+V6; (=3*V6)
0722	8AAE	VA=6*V6
0724	8AAE	VA=12*V6; (= Buffer pointer)
0726	00EE	Return
0728	6A10	VA=10; Set MSD equal to one
072A	F60A	Wait for Key Press
072C	6704	Test for Index<= 14 Hex
072E	8765	
0730	3F01	
0732	1738	Go to error message

Address	Code	Comments
0734	86A4	V6=V6+VA (=INDEX)
0736	1718	Go to 0718
0738	F629	Display LSD
073A	7405	Increment X Co-ordinate and
073C	D455	Display it
073E	6969	Message: "Index too large"
0740	2758	
0742	6880	Delay
0744	27a0	
0746	6962	Message: "Index="
0748	2750	
074A	1704	Go to 0704 and try again
074C	0000	Filler
074E	0000	
0750	00E0	Erase display
0752	6400	Set display co-ordinates to 00
0754	6500	" " " "
0756	175C	Go to 075C
0758	6400	Set X co-ord of display to 00 and
075A	7510	increment Y co-ord for next display line
075C	A600	Message buffer pointer
075E	F91E	
0760	F065	V0=ASCII Character
0762	7901	increment buffer; V9=V9+01
0764	4000	TEST FOR END of message (00 is EOM flag)
0766	00EE	Yes? Then return, otherwise
0768	2770	Go to character display subroutine
076A	175C	Get next character
076C	0000	Filler
076E	0000	"
0770	A400	Point to ASCII Code table
0772	F01E	Table pointer
0774	F065	V0=Pointer to display pattern
0776	A500	Point to character table pattern
0778	F01E	Pointer to display pattern
077A	D455	Display character
077C	7404	Increment X co-ord of display
077E	3440	Test for end of line
0780	00EE	Return
0782	7510	increment Y co-ord of display and
0784	6400	Set X co-ord to 00 for a new display line
0786	3580	Test for end of page
0788	00EE	Return
078A	00E0	Erase screen
078C	6400	Set X co-ord to 00 and
078E	6500	Set Y co-ord to 00
0790	00EE	Return
07A0	F815	Set Timer=V8
07A2	FD07	VD=Current Timer value
07A4	3D00	Test if Timer value=00
07A6	17A2	No? then Go to 07A2 and check it again
07A8	00EE	Yes? Then Return

BY [illegible] DATE 17 DEC 79 SUBJECT VIP AUTODIALER SHEET NO. ____ OF ____

CHKD. BY ____ DATE ____ JOB NO. W3FEY

DTMF ENCODER



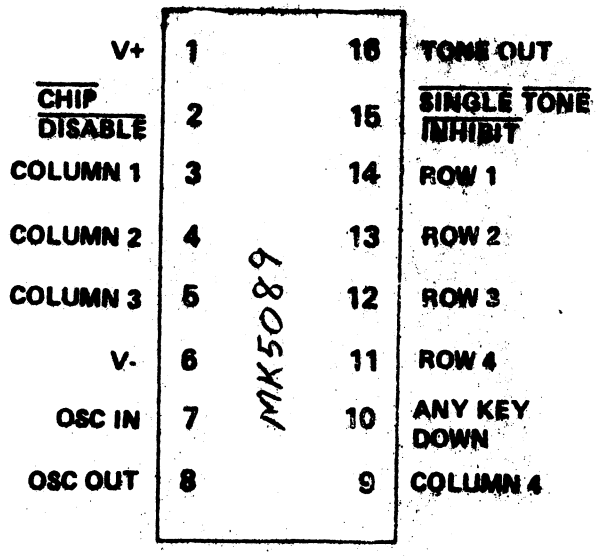


Fig. 1

3.05.31

DRIVING A BAUDOT PRINTER WITH THE VIP

BY GERALD KRIZEK

THIS PROGRAM IS AN ADAPTATION OF A PROGRAM PRESENTED BY B. MILLER IN IPSO FACTO #6, PAGE 38.

EVER SINCE I HAVE HAD BASIC OPERATING IN MY VIP, I HAD WISHED I HAD SOME SORT OF HARD COPY OUTPUT. UNFORTUNATLY, A DECENT PRINTER COSTS THE BETTER PART OF A KILOBUCK, MUCH MORE THAN MY HOBBY BUDGET WOULD ALLOW. OWNING A TELETYPE MODEL 15 FOR AMATEUR RADIO USE, I HAD BEEN GOING TO WRITE SOFTWARE TO USE IT WHEN I CAME ACROSS MILLER'S ARTICLE.

I HAVE ADAPTED THIS PROGRAM FOR USE ON A VIP RUNNING RCA'S VP-701 FLOATING POINT BASIC, HENCE THE LOCATION (0600-07FF).

SEVERAL MODIFICATIONS TO VP-701 ARE NECESSARY FOR THIS PROGRAM TO RUN.

FIRST: SINCE THE Q LINE IS USED FOR THE SERIAL OUTPUT, THE KEY DEPRESSED TONE MUST BE DEFEATED. DO THIS BY CHANGING LOCATION 2094 FROM 02 TO 00.

SECOND: THE INTERUPTS ASSOCIATED WITH THE TV OUTPUT REALLY FOUL UP THE TIMING FOR THE SERIAL OUTPUT. TO KEEP THE TV OFF PERMANENTLY, CHANGE LOCATION 1B6A FROM 98 TO D5.

THIRD: THE BASIC TABLE THAT CONTAINS THE ADDRESS OF THE OUTPUT ROUTINE MUST BE CHANGED. CHANGE LOCATIONS 2060 - 2061 FROM 08 - 00 TO 06 - 20.

THIS PROGRAM RECEIVES THE ASCII CHARACTER AND FIRST DETERMINES WHETHER IT IS A "SPACE", "LETTER", OR "FIGURE". IT THEN CHECKS THE LET/FIG MEMORY TO SEE WHAT THE LAST CHARACTER WAS AND DETERMINES IF IT SHOULD OUTPUT A "LETTERS" OR "FIGURES" FUNCTION BEFORE IT OUTPUTS THE BAUDOT VERSION OF THE ASCII INPUT.

IT THEN CHECKS THE LOOK-UP TABLE AT THE ASCII ADDRESS AND OUTPUTS THE DATA AT THAT LOCATION. THE OUTPUT IS MADE UP OF ONE START SPACE, FIVE INFORMATION BAUDS, AND TWO STOP MARKS.

THEN THE LETTER/FIGURE MEMORY IS UPDATED AND THE VIP RETURNS TO BASIC.

TIMING IS SET FOR A 60 WPM BAUDOT MACHINE. IF A HIGHER SPEED MACHINE IS USED, THE DATA AT LOCATION 06E2 WILL HAVE TO BE ADJUSTED FOR PROPER TIMING.

SINCE THE COMPUTER IS TIED UP WHILE OUTPUTING A CHARACTER, ANY INPUT AT THE ASCII KEYBOARD DURING EXECUTION OF THE OUTPUT ROUTINE WILL BE IGNORED.

SINCE A BAUDOT PRINTER DOES NOT HAVE SOME OF THE CHARACTERS IN THE ASCII CHARACTER SET, SOME SYMBOLS DUE DOUBLE-DUTY AS NOTED BY THE ASTERISK IN THE LOOK-UP TABLE LISTING. THIS HAS CAUSED ME NO PROBLEMS.

THE UNUSED MEMORY GAPS IN THIS PROGRAM ALLOWED ME TO EASILY MAKE CHANGES. SINCE I AM NOT PRESSED FOR MEMORY IN THIS AREA, I DID NOT BOTHER TO CONDENSE THE PROGRAM

I WILL BE HAPPY TO ANSWER ANY QUESTIONS ABOUT THIS PROGRAM THAT ARE MAILED TO ME. PLEASE SEND S.A.S.E.

GERALD KRIZEK
722 N. MORADA AVE.
WEST COVINA, CA.
91790

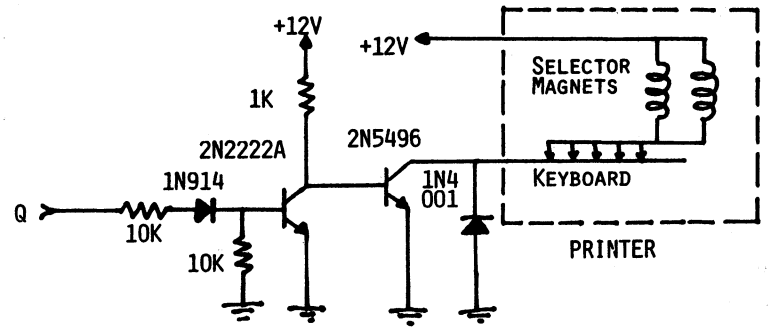
```
0600                        ; LETTER/FIGURE MEMORY LOCATION
ENTER
0620  22          DEC  R2   ;
  21  52          STR  R2   ; SAVE D
  22  22          DEC  R2   ;
  23  8E          GLO  RE   ; SAVE RE
  24  52          STR  R2   ;
  25  22          DEC  R2   ;
  26  9E          GHI  RE   ;
  27  52          STR  R2   ;
  28  F8 06       LDI  06   ; LOAD RF.1 WITH   LET/FIG MEM LOC
  2A  BF          PHI  RF   ;
  2B  F8 00       LDI  00   ;
  2D  AF          PLO  RF   ;
  2E  0F          LDN  RF   ;
  2F  BF          PHI  RF   ;
0630  12 12       INC  R2   ; RESTORE D
  32  02          LDN  R2   ;
  33  22 22 22    DEC  R2   ; DECREMENT STACK
  36  30 40       BR        ; BRANCH TO 0640
PROCESS ASCII
0640  AF          PLO  RF   ;
  41  FB 20       XRI  20   ; COMPARE ASCII TO SPACE
  43  32 9C       BZ   9C   ; IF SAME, GOTO 069C
  45  8F          GLO  RF   ;
  46  FE FE       SHL  SHL  ; SHIFT OUT 2 M.S.B.
  48  3B 90       BNF  90   ; IF FIGURE, GOTO 0690
  4A  9F          GHI  RF   ;
  4B  FB FF       XRI  FF   ; COMPARE ASCII TO LETTER
  4D  32 9C       BZ   9C   ; IF SAME, GOTO 069C
  4F  F8 81       LDI  81   ;
0651  D4 06 B0    CALL PRINT  : OUTPUT "LETTERS"
  54  30 9C       BR        ; GOTO 069C
  56  8F          GLO  RF   ;
  57  D4 06 B0    CALL PRINT  ;
  5A  F8 06       LDI  06   ; LOAD RF.1 WITH LET/FIG MEM LOC.
  5C  BF          PHI  RF   ;
  5D  F8 00       LDI  00   ;
  5F  AF          PLO  RF   ;
```

```
PROCESS ASCII CONT.
0660  F8 00       LDI  00   ; LOAD "FIG" INTO LET/FIG MEM.
  62  5F          STR  RF   ;
  63  30 80       BR        ; GOTO 0680
EXIT
0680  12          INC  R2   ; RESTORE RE
  81  02          LDN  R2   ;
  82  BE          PHI  RE   ;
  83  12          INC  R2   ;
  84  02          LDN  R2   ;
  85  AE          PLO  RE   ;
  86  12          INC  R2   ; RESTORE D
  87  02          LDN  R2   ;
  88  12          INC  R2   ;
  89  D5          SEP  R5   ; RETURN
PROCESS ASCII CONT.
0690  9F          GHI  RF   ; COMPARE ASCII TO FIGURE
  91  FB 00       XRI  00   ;
  93  32 56       BZ   56   ; IF SAME GOTO 0656
  95  F8 80       LDI  80   ; OUTPUT "FIGURES"
  97  D4 06 B0    CALL PRINT  ;
  9A  30 56       BR        ; GOTO 0656
  9C  8F          GLO  RF   ;
  9D  D4 06 B0    CALL PRINT  ; OUTPUT ASCII
06A0  F8 06       LDI  06   ; LOAD RF.1 WITH LET/FIG MEM LOC.
  A2  BF          PHI  RF   ;
  A3  F8 00       LDI  00   ;
  A5  AF          PLO  RF   ;
  A6  F8 FF       LDI  FF   ;
  A8  5F          STR  RF   ; LOAD "LET" INTO LET/FIG MEM.
  A9  30 80       BR        ; GOTO 0680
PRINT
06B0  AE          PLO  RE   ; ASCII TO RE.0
  B1  F8 07       LDI  07   ;
  B3  BE          PHI  RE   ; 07 TO RE.1
  B4  0E          LDN  RE   ; LOAD DATA FROM LOOK-UP TABLE
  B5  FE FE       SHL  SHL  ; SHIFT OUT 2 M.S.B.
  B7  7B          SEQ       ; START SPACE
  B8  D4 06 E0    CALL DELAY  ;
  BB  FE          SHL       ; SHIFT OUT M.S.B.
```

```
PRINT CONT.
06BC  32 C9       BZ          ; GOTO 06C9 IF D=0
  BE  3B C3       BNF         ; GOTO 06C3 IF DF=0
06C0  7A          REQ         ; SEND MARK
  C1  30 C4       BR          ; GOTO 06C4
  C3  7B          SEQ         ; SEND SPACE
  C4  D4 06 E0    CALL DELAY  ;
  C7  30 BB       BR          ; GOTO 06BB
  C9  7A          REQ         ; SEND STOP MARK
  CA  D4 06 E0    CALL DELAY  ;
  CD  D4 06 E0    CALL DELAY  ;
06D0  D5          SEP  R5     ; RETURN
DELAY
06E0  52          STR  R2     ; SAVE D
  E1  F8 FF       LDI  FF     ; DELAY CONSTANT
  E3  FF 01       SMI  01     ; D-1
  E5  C4 C4 C4    NO OP       ;
  E8  C4 C4       NO OP       ;
  EA  3A E3       BNZ  E3     ; GOTO 06E3 IF D≠0
  EC  02          LDN  R2     ; RESTORE D
  ED  D5          SEP  R5     ; RETURN
```

PRINTER INTERFACE

WITH A 12V SUPPLY AND THE SELECTOR MAGNETS IN PARALLEL THE LOOP CURRENT WILL BE APPROX. 60 MA.



ASCII TO BAUDOT LOOK-UP TABLE

ADDRESS	DATA	ASCII	BAUDOT
0700	37	NUL	*"FIGS"
07	29	BEL	BELL
0A	11	LF	LINE FEED
0D	05	CR	CARR. RET.
0713	37	DC-3	*"FIGS"
0720	09	SP	SPACE
21	2D	!	!
22	23	"	"
24	25	$	$
25	2F	%	*/
26	17	&	&
27	0D	'	'
28	3D	(	(
29	13	)	)
2A	0F	*	*.
2B	17	+	*&
2C	0D	,	,
2D	31	-	-
2E	0F	.	.
2F	2F	/	/
0730	1B	0	0
31	3B	1	1
32	33	2	2
33	21	3	3
34	15	4	4
35	03	5	5
36	2B	6	6
37	39	7	7
38	19	8	8
39	07	9	9
3A	1D	:	:
3B	1F	;	;
3C	3D	<	*(
3D	1D	=	*:
3E	13	>	*)
3F	27	?	?

ADDRESS	DATA	ASCII	BAUDOT
0740	17	a	&
41	31	A	A
42	27	B	B
43	1D	C	C
44	25	D	D
45	21	E	E
46	2D	F	F
47	17	G	G
48	0B	H	H
49	19	I	I
4A	35	J	J
4B	3D	K	K
4C	13	L	L
4D	0F	M	M
4E	0D	N	N
4F	07	O	O
0750	1B	P	P
51	3B	Q	Q
52	15	R	R
53	29	S	S
54	03	T	T
55	39	U	U
56	1F	V	V
57	33	W	W
58	2F	X	X
59	2B	Y	Y
5A	23	Z	Z
0780	37		"FIGS"
81	3F		"LETTERS"

ANY ADDRESS BETWEEN 0700 AND 07FF THAT IS NOT SHOWN ABOVE SHOULD CONTAIN 01 AS DATA. BAUDOT CHARACTERS MARKED (*) WERE CHOSEN TO REPRESENT ASCII CHARACTERS FOR WHICH THERE IS NO BAUDOT EQUIVALENT.

VIP Hobby Computer Association
32 Ainsworth Avenue
East Brunswick, NJ 08816

A FINAL WORD:

This issue of VIPER is the largest to date put out by VIPHCA. I hope that you will enjoy it and find the material worthwhile. It really is a big help when you can send in material that can be just "pasted up". Speaking of material, Richard D. Ward sent in a CHIP-8 version of the game of WAMPUS which runs in the 4K VIP. But his documentation, which is superb, is over 38 pages!!! Perhaps we might consider having a special publication of just that game for those who are interested. Let me know what you think. Or perhaps I should just print it in parts as a part of the VIPER.

Let me wish all of you a happy Holiday Season and a worthwhile 1982! After this issue of VIPER, there will be one more for the "1981" membership year. I have several good articles in the can, including a Tom Swan game, but if you have anything special you might like to send in for #6, please do! (RS)